High-Level Synthesis Implementation of an Accurate HEVC Interpolation Filter on an FPGA

Panu Sjövall, Matti Rasinen, Ari Lemmetti, Jarno Vanne
Ultra Video Group, Tampere University, Finland
{panu.sjovall, matti.rasinen, ari.lemmetti, jarno.vanne}@tuni.fi

***Abstract*—This paper presents the first known high-level synthesis (HLS) implementation of an accurate interpolation filter for High Efficiency Video Coding (HEVC). The proposed multiplierless shift-register-based architecture is able to interpolate effectively up to four 8×8 blocks at a time for HEVC fractional motion estimation (FME). Our filter is implemented on Intel Arria V and Xilinx Virtex 6 FPGAs. On Arria V, it can operate at 270 MHz with 21.1 kALUTs. According to our profiling results, it can filter an adequate number of samples for FME in real-time 4K HEVC encoding of up to 85 frames per second (fps). On Virtex 6, the respective values are 313 MHz, 27.1 kLUTs, and 99 fps. The proposed solution doubles the speed over any of the existing interpolation filters for HEVC FME on an FPGA. It is also the only interpolation filter that meets the needs of real-time 4K HEVC encoder in practice and without any compromises in 23-bit filtering accuracy**.

Keywords—High Efficiency Video Coding (HEVC), fractional motion estimation (FME), interpolation filter, high-level synthesis (HLS), field-programmable gate array (FPGA)

I. Introduction

High Efficiency Video Coding (*HEVC/H.265*) [1], [2], is the latest widespread international video coding standard. It is able to reduce the bitrate by almost 50% for the same visual quality over the preceding *Advanced Video Coding* (*AVC/H.264*) standard. HEVC coding gain stems mainly from the new block partitioning structure and improved *motion compensated prediction* (*MCP*) that is used to remove temporal redundancy between video frames.

The interpolation filter is a normative coding tool of MCP. It interpolates samples between integer pixels in fractional-pixel precision for *fractional motion estimation* (*FME*) and fractional-pixel sampling in general. In HEVC, the accuracy of this filter was improved from that of AVC by refining the filter coefficients, using longer filter taps, and increasing the precision of filtering operations [3]. However, higher accuracy also introduces additional complexity. For example, the HEVC interpolation was reported to account for up to 38% of the whole encoding complexity in *HEVC test model* (*HM*) [4]. In software implementations, the time used for filtering can be reduced by multithreading and vectorization [5]. Further speedup and lower power budget are typically sought with dedicated hardware accelerators [6]-[11].

This work proposes a real-time HEVC interpolation filter implementation on a *field-programmable gate array* (*FPGA*). It is designed to filter luma samples for FME in a practical HEVC encoder. As in [12], our proposal is sped up by interpolating samples for 9×9-sized blocks at a time instead of producing the respective samples by filtering four overlapping 8×8 blocks individually.

The proposed filter was written in C/C++ at behavioral level and synthesized for an FPGA with *High-level synthesis* (*HLS*) [13] tool called Catapult [14]. Catapult HLS tool can automatically generate *register-transfer-level* (*RTL*) description from C/C++ without the need to rewrite the code with traditional *hardware description languages* (*HDLs*) like VHDL and Verilog. Increasing the design abstraction from RTL to behavioral level has been reported to provide 4-6 times increase in productivity [15], mainly because the behavioral code is more readable, design and verification times are shorter, and the design reusability is far better than with handwritten HDL equivalents. All these aspects motivated us to implement the proposed interpolation filter with an HLS tool. To the best of our knowledge, this is also the first HLS implementation for an accurate HEVC interpolation filter.

The rest of the paper is organized as follows. Section II addresses the related FPGA architectures for HEVC interpolation. Section III describes the HEVC interpolation algorithm and section IV introduces the proposed filtering scheme and architecture for it. In addition, the implementation aspects of HLS are considered. Section V compares the performance and resource consumption of the proposed system over the related work on FPGAs and evaluates their feasibility for real-time 4K HEVC encoding. Section VI concludes the paper.

II. Related Work

The majority of the HEVC interpolation filter complexity comes from *multiply and accumulate* (*MAC*) operations. In the existing FPGA filter architectures, the filtering complexity was primarily mitigated by reducing the number of costly multiplications in MAC operations [9] or eliminating them completely [6]-[8], [10], [11].

Another common optimization approach is to implement an approximation of the standard 7/8-tap HEVC interpolation filter that requires 16-bit input, 23-bit intermediate, and 17-bit output values for full-precision interpolation. The approximations were carried out by reducing bit widths of intermediate interpolation values [7], [8], [10], [11], simplifying filtering coefficients [7], decreasing the number of filter taps [7], or reusing the filters [8], [10], [11]. A common drawback of all these techniques is their negative effect on coding efficiency over the full-precision designs [6], [9].

Thirdly, majority of these existing works used either a 15×15 pixel [6], [7], [10], [11] or a 16×16 pixel [8], [9] reference array to interpolate 8×8 output samples for FME. However, as introduced in [12] interpolating 9×9 samples from a 16×16-pixel array can be as efficient as four separately interpolated 8×8 blocks.

TABLE I. FILTER COEFFICIENTS

Index i	-3	-2	-1	0	1	2	3	4
qfilter	-1	4	-10	58	17	-5	1	
hfilter	-1	4	-11	40	40	-11	4	-1
qfilter^{-1}		1	-5	17	58	-10	4	-1

Finally, only one [11] of these prior works was implemented with HLS. Hence, our proposal is the second known HLS implementation for HEVC interpolation and the first HLS solution that operates at real-time and at full accuracy.

III. FRACTIONAL SAMPLE INTERPOLATION ALGORITHM

The HEVC standard [1] specifies three sets of filter coefficients for 7/8-tap luma sample interpolation. Each of them corresponds to a quarter-pixel displacement: +1/4 px, +2/4 px, and +3/4 px. Here, they are respectively classified as *qfilter*, *hfilter*, and *qfilter^{-1}* coefficients, as tabulated in Table 1. The same weights are used for both the horizontal and vertical steps of the filtering. Furthermore, the *qfilter* and *qfilter^{-1}* coefficients are made up of the same seven weights, but in reverse order.

Fig. 1 illustrates the HEVC interpolation scheme for luminance samples. The *qfilter* coefficients are used for calculating samples $a_{x,y}$, $d_{x,y}$, $e_{x,y}$, $f_{x,y}$, $g_{x,y}$, $i_{x,y}$, $p_{x,y}$, the *hfilter* coefficients for samples $b_{x,y}$, $f_{x,y}$, $h_{x,y}$, $i_{x,y}$, $j_{x,y}$, $k_{x,y}$, $q_{x,y}$, and the *qfilter^{-1}* coefficients for samples $c_{x,y}$, $g_{x,y}$, $k_{x,y}$, $n_{x,y}$, $p_{x,y}$, $q_{x,y}$, $r_{x,y}$. In Fig. 1, they are denoted as red, violet, and black tags, respectively.

Fig. 1 also shows the decomposition of the filtering process into horizontal, vertical, and diagonal cases based on the HEVC specification [1]. For example, samples $a_{0,0}$, $b_{0,0}$, and $c_{0,0}$ are horizontally filtered from integer pixels $A_{-3,0}$ to $A_{4,0}$ as

$$a_{0,0} = \left(\sum_{x=-3}^{3} A_{x,0} \times qfilter[x]\right) \gg (B-8),$$

$$b_{0,0} = \left(\sum_{x=-3}^{4} A_{x,0} \times hfilter[x]\right) \gg (B-8), \text{ and}$$

$$c_{0,0} = \left(\sum_{x=-2}^{4} A_{x,0} \times qfilter^{-1}[x]\right) \gg (B-8),$$

where >> denotes right shift and B equals the bit depth of the reference samples. Vertical filtering is conducted accordingly but with integer pixels in the vertical direction. For example, $d_{0,0}$ is filtered from integer pixels $A_{0,-3}$ to $A_{0,3}$.

Furthermore, diagonal samples are interpolated from the output samples of the horizontal filtering in the vertical direction. For example, samples $e_{0,0}$, $i_{0,0}$, and $p_{0,0}$ are interpolated from $a_{x,y}$ samples as

$$e_{0,0} = \left(\sum_{y=-3}^{3} a_{0,y} \times qfilter[y]\right) \gg 6,$$

$$i_{0,0} = \left(\sum_{y=-3}^{4} a_{0,y} \times hfilter[y]\right) \gg 6, \text{ and}$$

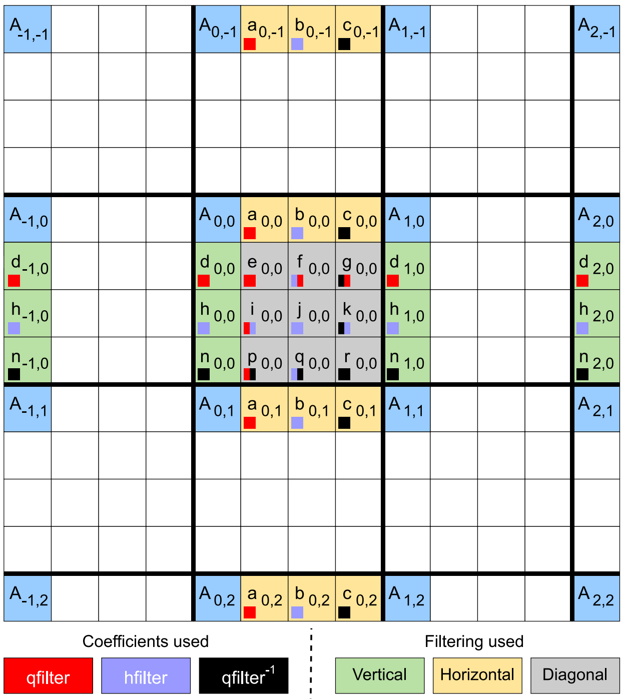

Fig. 1. Luma integer pixel, ½ pixel, and ¼ pixel positioning.

$$p_{0,0} = \left(\sum_{y=-2}^{4} a_{0,y} \times qfilter^{-1}[y]\right) \gg 6.$$

Samples $f_{x,y}$, $j_{x,y}$, and $q_{x,y}$ follow the same formula but are interpolated from adjacent $b_{x,y}$ samples. Respectively, samples $g_{x,y}$, $k_{x,y}$, and $r_{x,y}$ are filtered from adjacent $c_{x,y}$ samples. Every sample that goes through the filtering process is finally rounded and clipped to input bit width [1].

IV. PROPOSED INTERPOLATION FILTER ARCHITECTURE

The proposed hardware architecture is a novel combination of the following four implementation techniques:

1) A multiplierless filtering as in [6]-[8], [10], [11];
2) Shift-register-based filtering as in [9];
3) Simultaneous interpolation of 9×9 samples as in [12];
4) HLS design flow as in [11];

A. Multiplierless Filtering

In the multiplierless filtering algorithm, multiplications are replaced with shift/adder structures. For example, the original horizontal filtering operation for a sample (S) is computed as

$$S = \begin{pmatrix} -a_0 + 4 \times a_1 - 10 \times a_2 + 58 \times a_3 \\ +17 \times a_4 - 5 \times a_5 + a_6 \end{pmatrix} \gg Shift \quad (1)$$

Altogether, it contains five multiplications with *qfilter* coefficients. Here the sample indices refer to x or y coordinate depending on the filtering direction. Replacing the multiplications with shifting and adding yields

$$S = \begin{bmatrix} -a_0 + a_6 - a_5 + a_4 + a_3 \ll 5 \\ +(a_3 - a_2) \ll 3 + (a_1 - a_5) \ll 2 \\ +(a_3 - a_2) \ll 1 + (a_3 + a_4) \ll 4 \end{bmatrix} \gg Shift. \quad (2)$$

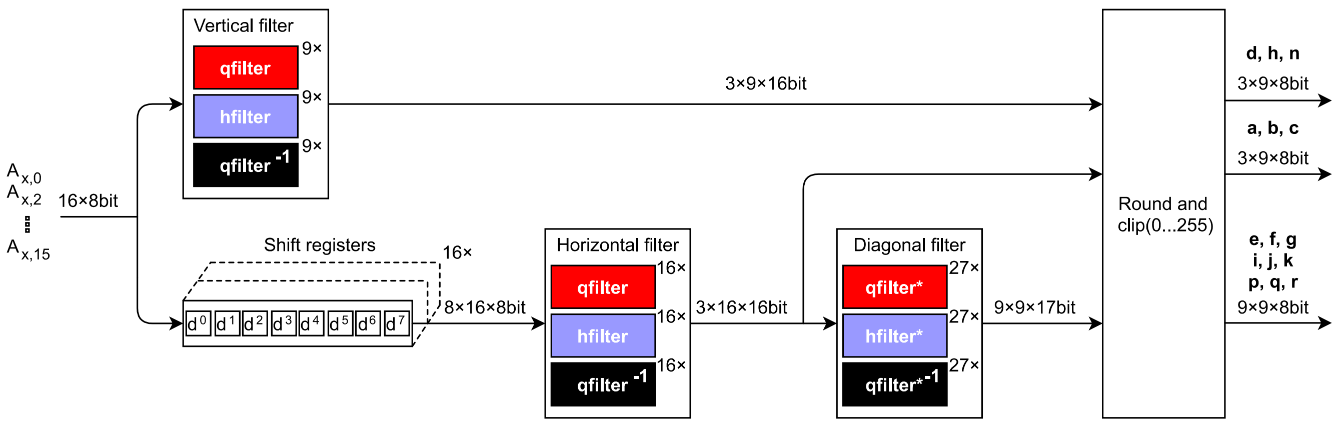

Fig. 2. The proposed multiplierless shift-register-based architecture for HEVC interpolation (*16-bit input, shifting according to (3)).

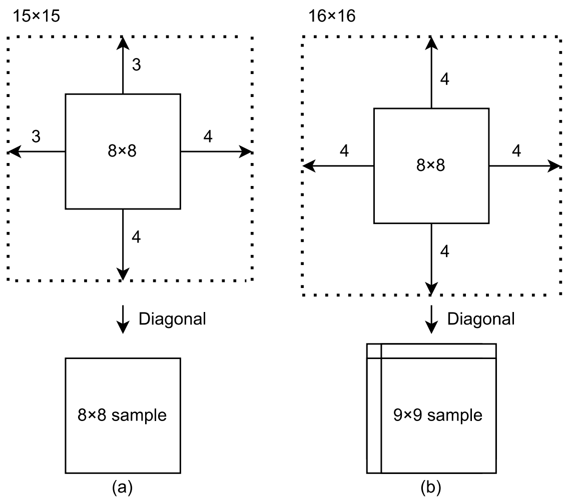

Fig. 3. Interpolation of diagonal samples. (a) 8×8. (b) 9×9.

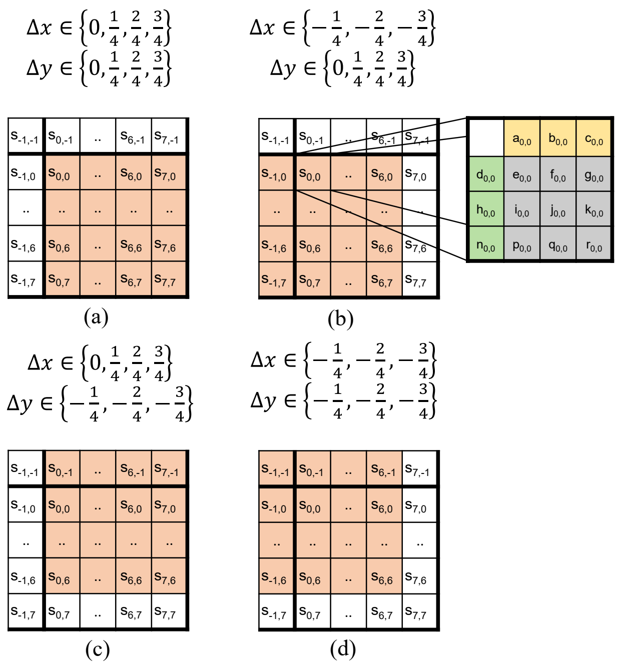
$\Delta x \in \left\{0, \frac{1}{4}, \frac{2}{4}, \frac{3}{4}\right\}$
$\Delta y \in \left\{0, \frac{1}{4}, \frac{2}{4}, \frac{3}{4}\right\}$
$\Delta x \in \left\{-\frac{1}{4}, -\frac{2}{4}, -\frac{3}{4}\right\}$
$\Delta y \in \left\{0, \frac{1}{4}, \frac{2}{4}, \frac{3}{4}\right\}$
$s_{-1,-1}$ $s_{0,-1}$ .. $s_{6,-1}$ $s_{7,-1}$
$s_{-1,0}$ $s_{0,0}$ .. $s_{6,0}$ $s_{7,0}$
$s_{-1,6}$ $s_{0,6}$ .. $s_{6,6}$ $s_{7,6}$
$s_{-1,7}$ $s_{0,7}$ .. $s_{6,7}$ $s_{7,7}$
$a_{0,0}$ $b_{0,0}$ $c_{0,0}$
$d_{0,0}$ $e_{0,0}$ $f_{0,0}$ $g_{0,0}$
$h_{0,0}$ $i_{0,0}$ $j_{0,0}$ $k_{0,0}$
$n_{0,0}$ $p_{0,0}$ $q_{0,0}$ $r_{0,0}$
(a)
(b)
$\Delta x \in \left\{0, \frac{1}{4}, \frac{2}{4}, \frac{3}{4}\right\}$
$\Delta y \in \left\{-\frac{1}{4}, -\frac{2}{4}, -\frac{3}{4}\right\}$
$\Delta x \in \left\{-\frac{1}{4}, -\frac{2}{4}, -\frac{3}{4}\right\}$
$\Delta y \in \left\{-\frac{1}{4}, -\frac{2}{4}, -\frac{3}{4}\right\}$
(c)
(d)

Fig. 4. Fractional samples required by FME. (a) Initial. (b) Left. (c) Top. (d) Top-left.

In (1) and (2), *Shift* depends on the filter (see Fig. 1) so that

$$Shift = \begin{cases} B-8, & filter \in \{horizontal, vertical\} \\ 6, & filter \in \{diagonal\} \end{cases}. \tag{3}$$

This ensures that shifting is done without any loss in accuracy over the HEVC specification [1].

The multiplierless filtering approach reduces the hardware cost of the interpolation engine significantly. In our case, Catapult was applied to synthesize both (1) and (2) for Arria V, and the area estimate for (2) was only 2.5% of that of (1) with equivalent synthesis tool constraints.

B. *Shift-Register-Based Filtering*

Fig. 2 depicts our shift-register based filtering architecture adopted from [9]. It is composed of vertical, horizontal, and diagonal filters, shift register with a depth of 16 pixels, and a round and clip unit. Contrary to [9], the proposed implementation does not need any multipliers but it uses parallel 8×16-shift-registers to feed the horizontal filter. This method requires that integer pixels are received column by column. For example, the first column includes pixels $A_{-4,-4}$, $A_{-4,-3}$, … $A_{-4,12}$. Both horizontal and diagonal filters have internal latencies of 8 cycles whereas the latency of the vertical filter is only 6 clock cycles. After initial latency of 23 clock cycles, architecture can produce 15×9×9 samples every 16 clock cycles.

The horizontal filter has 8×16=128 8-bit inputs and 3×16=48 16-bit outputs. It can produce a column of horizontally filtered samples for each fractional-pixel offset $a_{x,y}$, $b_{x,y}$, and $c_{x,y}$ (3×9 samples) per clock cycle. The vertical filter has the same throughput, but it filters sample columns ($d_{x,y}$, $h_{x,y}$, $n_{x,y}$) vertically without utilizing shift-registers. The diagonal filter produces the rest of the sample columns at the rate of 9 columns (9×9 samples) per clock cycle by applying vertical filtering to horizontally filtered samples. The diagonal filter uses 16-bits for the input and 17-bits for the output.

The intermediate values may require maximum of 23 bits in extreme cases if the offsetting method is not used to avoid possible overflow [3]. Right shifting the values by 6, according to (3), makes it necessary to have a 17-bit output to preserve the required full-precision. All samples are finally rounded and clipped.

C. *Simultaneous Interpolation of 9x9 Samples*

Interpolating an 8×8 block of samples with HEVC-compliant 7/8-tap filters requires an array of 15×15 reference integer pixels. Therefore, the reference sample block of 8×8 integer pixels needs to be extended by 3 extra integer pixels left and upward as well as 4 extra integer pixels right and downward as depicted in Fig. 3(a).

In FME, the best adjacent fractional-pixel offset can be found in any direction given by vertical (Δy) and horizontal (Δx) quarter-pixel offsets around the sample position of interest. Since the filter coefficients are exclusively defined for positive fractional offsets, only the pixels with fractional offsets to the right and downwards can be completely interpolated from an array of 15×15 reference pixels. This is illustrated in Fig. 4(a) with a shaded 8×8 block containing all interpolated samples.

Instead, negative fractional offsets are derived by adding positive fractional offsets to negative integer offsets, such as $\Delta x = -1/4 = -1 + 3/4$. Thus, performing FME in the remaining directions requires that the interpolation window is moved left, up, and diagonally up-left by one pixel to be able to interpolate all the required samples, as shown in Fig. 4(b)-(d), respectively. At sample level, 15 fractional positions (*a*-*r*) per interpolation window position need to be filtered around an integer sample. For an 8×8 block, it means that 4 × 15 × 8 × 8 = 3840 fractional-pixels (*s*) are interpolated [6], [7], [10], [11].

As illustrated by Fig. 3(b), extending the reference array by one extra pixel left and upward to 16×16 pixels allows for more efficient interpolation for FME [12] since it eliminates redundant filtering needed in the traditional approach. The majority of the fractional samples are anyway interpolated regardless of the interpolation window position. When the left and above edges of a block are also interpolated, producing blocks of 9×9 samples for each fractional-pixel position (*a*-*r*) at a time provides all quarter-pixel (and half-pixel) samples required for FME with a single block interpolation pass.

With this minimal increase to the reference array and resources, the interpolation calls for FME can be reduced from 25 to 16 for 32×32 blocks, 9 to 4 for 16×16 blocks, and 4 to 1 for 8×8 blocks.

D. *HLS Design Aspects*

The C implementation of the proposed interpolation filter was originally designed for our open-source Kvazaar HEVC encoder [16] and then further optimized for synthesis with Catapult HLS tool. The HLS approach sped up the implementation phase considerably because the RTL description was automatically generated from the C code without any manual design phases for HDL description. The HLS code for the vertical filter is listed in Appendix A as an example.

Catapult calculates and modifies the throughput of the design based on the target technology, desired clock frequency, and other architectural constraints. It can automatically adjust the number of combinatorial operations per clock cycle and pipeline stages of the state machine for the desired throughput. For example, our filter design was optimized for speed, so Catapult was configured to unroll all nested loops for the optimal throughput. These loop settings could have easily been modified for reduced area, e.g., for a smaller FPGA.

For faster verification, Catapult also uses the same C source code for automatic testbench generation. The testbench synchronizes with the input and output of the design under verification, so it is tolerant of architectural changes. In this work, the same testbench was used along the whole design process including behavioral functionality verification, RTL simulation, and FPGA prototyping.

HLS also offers better design reusability over traditional design approaches. A technology-independent behavioral code releases the designers from addressing the implementation details of the target technology, such as timing, interfaces, and memory elements. In principle, the same holds for the handwritten RTL code but the design is usually implemented with a specific technology and performance in mind. In this work, it was straightforward to generate optimized RTL designs for both Intel and Xilinx FPGAs, by only changing the target technology in Catapult.

V. Performance Analysis

Table 2 reports the area and performance results of the proposed interpolation architecture on Intel Arria V and Xilinx Virtex 6 FPGAs together with the prior art. Our proposal was synthesized with Intel Quartus Prime 18.1 for Arria V and Mentor Precision Synthesis 2019.2.0.9 for Virtex 6. The obtained results were benchmarked against related HEVC interpolation implementations on FPGA.

A. *Resource and Throughput Evaluation*

On Arria V, the proposed architecture can operate at 270 MHz frequency and it uses 18.9 kALUTs with the 15×15 input array of reference pixels. Altogether, 15 interpolated samples are filtered per integer pixel, meaning that a total of 15 × (8 × 8) = 960 samples are interpolated per 8×8 block. The architecture completes a single 8×8 sample block in 15 cycles so it can output samples at a rate of (270 MHz × 960 samples) / 15 = 17 280 Msamples/s.

Expanding the size of the input array from 15×15 to 16×16 pixels increases the resource consumption by 11.3% to 21.1 kALUTs and latency to 16 cycles. With this minimal overhead, our proposal can be made to output fully interpolated 9×9 sample blocks, and the throughput increases by 18.7% to (270 MHz × 15 × (9 × 9) samples) / 16 = 20 503 Msamples/s. The respective results on Virtex 6 are 27.1 kLUTs, 313 MHz, and 23 786 Msamples/s.

The characteristics of the proposed implementation are superior to those of existing approaches [6]-[11]. The full-precision architectures presented by Pastuszak [6] et al. and Lung et al. [9] needed 36% and 5% more resources for 58% and 80% lower throughputs than ours, respectively. The rest of the solutions [7], [8], [10], [11] only implement approximated interpolation functionality that reduces accuracy over that of standard-compliant solutions, limit their usage, and degrades coding efficiency.

TABLE II. PERFORMANCE AND AREA RESULTS OF THE PROPOSED AND RELATED HEVC INTERPOLATION FILTER ARCHITECTURES ON FPGA

Architecture	FPGA	Logic Cells	Freq.	Input	Output	Cycles	Msamples/s	Speed (2160p)
Proposed (HLS)	Arria V	18.9 kALUT	270 MHz	15×15	8×8	15	17 280	49 fps
Proposed (HLS)	Arria V	21.1 kALUT	270 MHz	16×16	9×9	16	20 503	85 fps
Pastuszak[6]	Arria II	28.8 kALUT	200 MHz	15×15	8×8	15	12 800	36 fps
Penny [7]	Stratix V	5.7 kALUT	239 MHz	15×15	8×8	15	15 282	43 fps
Silva [8]	Cyclone 4	4.6 kALUT	76 MHz	16×16	8×8	47	1 562	4 fps
Proposed (HLS)	Virtex 6	27.1 kLUT	313 MHz	16×16	9×9	16	23 768	99 fps
Lung [9]	Virtex 5	28.5 kLUT	120 MHz	16×16	8×8	16	7 200	20 fps
Mert [10]	Virtex 6	3.8 kLUT	233 MHz	15×15	8×8	50	4 474	13 fps
Ghani [11] (HLS)	Virtex 6	14.2 kLUT	168 MHz	15×15	8×8	29	5 561	16 fps

TABLE III. KVAZAAR PARAMETERS IN OUR EXPERIMENTS

Feature	Kvazaar configuration
Preset	Ultrafast
Inter Blocks	32×32 / 16×16
Intra Blocks	16×16 / 8×8
Fractional-pixel ME	1/4-pixel precision*
Biprediction	Disabled*
Quantization Parameter	22
GOP Structure	Low Delay P, 4 frames* ("--gop=lp-g4d3t1")

* overrides the preset default

B. *Performance Estimation in Practical HEVC Encoder*

The effective throughputs of the proposed and existing solutions were also estimated by accommodating them to the needs of FME in our practical Kvazaar open-source HEVC encoder [5], [16] that was configured for real-time low-delay HEVC encoding.

Table 3 details the coding parameters of Kvazaar. The test set was made up of the eight 2160p (2160×3840 pixels) test video sequences (*CityAlley*, *FlowerFocus*, *FlowerKids*, *FlowerPan*, *RaceNight*, *RiverBank*, *SunBath*, and *Twilight*) from our UVG Dataset [17]. Our profiling was carried out by encoding the sequences at 50 *frames per second* (*fps*) to estimate the requirements of FME processing in real-time HEVC coding.

According to our profiling, Kvazaar FME required samples at average throughput of 17 719 Msamples/s with 8×8 sample blocks. However, increasing the sample block size to 9×9 reduced the throughput need by 32% to 12 055 Msamples/s if the extra interpolated area was properly used in FME.

On Arria V, our solution is able to output 8×8 sample blocks for Kvazaar FME at an adequate rate up to coding speed of 49 fps. Furthermore, increasing the sample block to 9×9 makes it possible to accelerate coding speed by 1.7× up to 85 fps. The higher operating frequency on Virtex 6 increases the frame rate up to 99 fps. These two solutions are 2.0× and 2.3× as fast as any of the existing solutions. Hence, only the proposed solutions make it possible to encode 2160p sequence in real time.

VI. CONCLUSION

This paper presented the first known HLS implementation of an accurate HEVC interpolation filter on FPGA. The C source code of the filter was originally designed for the Kvazaar HEVC encoder, then optimized for the Catapult HLS tool, and finally synthesized for Arria V and Virtex 6 FPGAs. Our multiplierless shift-register-based architecture can interpolate 9×9 samples from a 16×16 reference input and thereby yield four interpolated 8×8 blocks at a time for HEVC FME. This scheme was shown to almost double the speed of HEVC interpolation with a slight FPGA resource overhead.

According to our profiling results, the proposed two implementations are able to filter adequate number of samples for FME processing in practical 4K HEVC encoding up to frame rates of 85 and 99 fps, respectively. Our approach almost doubles the speed over any of the existing HEVC interpolation filters on FPGA. It also uses 23-bit intermediate results to preserve accuracy with all possible input pixel blocks. Hence, it is the only FPGA solution that meets the needs of real-time 4K HEVC encoder in practice and without any compromises in filtering accuracy. Furthermore, the emerging *Versatile Video Coding* (*VVC/H.266*) [18] standard specifies interpolation filters similar to those of HEVC, so the proposed techniques are suitable to accelerate VVC coding as well.

ACKNOWLEDGMENT

This paper is part of the ADACORSA project that has received funding within the ECSEL JU in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National Authorities, under grant agreement 876019.

REFERENCES

[1] *High Efficiency Video Coding, document ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC),* ITU-T and ISO/IEC, Nov. 2019.

[2] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1649-1668.

[3] K. Ugur, A. Alshin, E. Alshina, F. Bossen, W. Han, J. Park, and J. Lainema, "Motion compensated prediction and interpolation filter design in H.265/HEVC," *IEEE J. Sel. Topics Signal Process.*, vol. 7, no. 6, Dec. 2013, pp. 946-956

[4] J. Vanne, M. Viitanen, T. D. Hämäläinen, and A. Hallapuro, "Comparative rate-distortion-complexity analysis of HEVC and AVC video codecs," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1885-1898.

[5] A. Lemmetti, M. Viitanen, A. Mercat, and J. Vanne, "Kvazaar 2.0: fast and efficient open-source HEVC inter encoder," *in Proc. ACM Multimedia Syst. Conf.*, Istanbul, Turkey, June 2020.

[6] G. Pastuszak and M. Trochimiuk, "Architecture design of the high-throughput compensator and interpolator for the H.265/HEVC

encoder," *J. Real-Time Image Process.* vol. 11, no. 4, Apr. 2014, pp. 663-673.

[7] W. Penny, G. Correa, L. Agostini, D. Palomino, M. Porto, G. Nazar, and B. Zatt, "Low-power and memory-aware approximate hardware architecture for fractional motion estimation interpolation on HEVC," *in Proc. IEEE Int. Symp. Circuits Syst.*, Sevilla, Spain, Oct. 2020.

[8] R. da Silva, Í. Siqueira, and M. Grellert, "Approximate interpolation filters for the fractional motion estimation in HEVC encoders and their VLSI design," *in Proc. Symp. Integ.r Circuits Syst. Des.*, Sao Paulo, Brazil, Aug. 2019.

[9] C. Lung and C. Shen, "A high-throughput interpolator for fractional motion estimation in high efficient video coding (HEVC) systems," *in Proc. IEEE Asia Pacific Conf. Circuits Syst.*, Ishigaki, Japan, Nov. 2014.

[10] A. C. Mert, E. Kalali, and I. Hamzaoglu, "An HEVC fractional interpolation hardware using memory based constant multiplication," *in Proc. IEEE Int. Conf. Consum. Electron.*, Las Vegas, Nevada, USA, Jan. 2018,

[11] F. A. Ghani, E. Kalali, and I. Hamzaoglu, "FPGA implementations of HEVC sub-pixel interpolation using high-level synthesis," *in Proc. Int. Conf. Des. Technol. Integr. Syst. Nanoscale Era*, Istanbul, Turkey, Apr. 2016.

[12] G. Pastuszak and M. Trochimiuk, "Algorithm and architecture design of the motion estimation for the H.265/HEVC 4K-UHD encoder," *J. Real-Time Image Process.*, vol. 12, July 2015, pp. 663-673.

[13] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test Comput.*, vol. 26, no. 4, July-Aug. 2009, pp. 8-17.

[14] Catapult: HLS-verification [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/hls-verification

[15] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 38, no. 5, May 2019, pp. 898-911.

[16] *Kvazaar HEVC encoder.* [Online]. Available: https://github.com/ultravideo/kvazaar

[17] A. Mercat, M. Viitanen, and J. Vanne, "UVG dataset: 50/120fps 4K sequences for video codec analysis and development," *Proc. ACM Multimedia Syst. Conf.*, Istanbul, Turkey, June 2020.

[18] *Versatile Video Coding, Recommendation ITU-T Rec. H.266 and ISO/IEC 23090-3 (VVC),* ITU-T and ISO/IEC JTC 1, July 2020.

APPENDIX A

```
// Template data structures for passing data through channels
template<int width> struct IntegerPixel {
 ac_int<width,false> column[16];
};

template<int width, int filters, int samples> struct Filtered {
 ac_int<width,true> pixel[filters][samples];
};

// Template functions for the three filter coefficients: qfilter, hfilter, and inverse qfilter
template<int width_in, int width_out>
ac_int<width_out,true> qfilter(ac_int<width_in, false> a[8]) {
 ac_int<width_in,true> S = -a[0] + a[6] - a[5] + a[4] + (a[3] << 5)
                           + ((-a[2] + a[3]) << 3) + ((a[1] - a[5]) << 2)
                           + ((a[3] - a[2]) << 1) + ((a[3] + a[4]) << 4);
 return S;
}

template<int width_in, int width_out>
ac_int<width_out,true> hfilter(ac_int<width_in, false> a[8]) {
 ac_int<width_in,true> S = -a[0] - a[7] - a[2] - a[5] - ((a[2] + a[5]) << 1)
                           + ((a[1] + a[6]) << 2) + ((a[3] + a[4] - a[2] - a[5]) << 3)
                           + ((a[3] + a[4]) << 5);
 return S;
}

template<int width_in, int width_out>
ac_int<width_out,true> qfilter_1(ac_int<width_in, false> a[8]) {
 ac_int<width_in,true> S = -a[7] + a[1] - a[2] + a[3] + ((-a[5] + a[4]) << 3)
                           + ((a[6] - a[2]) << 2) + ((a[4] - a[5]) << 1)
                           + ((a[4] + a[3]) << 4) + (a[4] << 5);
 return S;
}

// Hierarchical unit for vertical filtering using qfilter, hfilter, and inverse qfilter
void vertical_filter(ac_channel< IntegerPixel<BIT_DEPTH> > &pixels,
                     ac_channel<Filtered<BIT_DEPTH*2, 3, 9> > &filt_ver_full) {
 IntegerPixel<BIT_DEPTH> pixels_in;
 Filtered<BIT_DEPTH*2, 3, 9> filt_out;
 pixels_in = pixels.read();
 #pragma hls_unroll yes
 for(uint4 x=0;x<9;x++) {
   filt_out.pixel[0][x] = qfilter<BIT_DEPTH,BIT_DEPTH*2>(pixels_in.column + x);
   filt_out.pixel[1][x] = hfilter<BIT_DEPTH,BIT_DEPTH*2>(pixels_in.column + x);
   filt_out.pixel[2][x] = qfilter_1<BIT_DEPTH,BIT_DEPTH*2>(pixels_in.column + x);
 }
 filt_ver_full.write(filt_out);
}

// Hierarchical unit for rounding and clipping
void ver_round_clip(ac_channel<Filtered<BIT_DEPTH*2, 3, 9> > &filt_ver_full,
                    ac_channel<Filtered<BIT_DEPTH, 3, 9> > &filt_ver_final){
 Filtered<BIT_DEPTH*2, 3, 9> filt_in = filt_ver_full.read();
 Filtered<BIT_DEPTH,   3, 9> filt_round_clip_out;
 #pragma hls_unroll yes
 for(uint3 a = 0; a < 3; a++) {
  #pragma hls_unroll yes
  for(uint4 b = 0; b < 9; b++) {
   // #define CLIP(low,high,value) MAX((low),MIN((high),(value)))
   filt_round_clip_out.pixel[a][b] =
   CLIP(PIXEL_MIN,  PIXEL_MAX, ((filt_in.pixel[a][b] + round) >> shift));
  }
 }
 filt_ver_final.write(filt_round_clip_out);
}

// Top-level unit for connecting the two hierarchical units
void vertical_top(ac_channel<IntegerPixel<BIT_DEPTH> > &pixels,
                  ac_channel<Filtered<BIT_DEPTH, 3, 9> > &filt_vert_final) {
 static ac_channel<Filtered<BIT_DEPTH*2, 3, 9> > filt_ver_hier;
 vertical_filter(pixels, filt_ver_hier);
 vertical_round_clip(filt_ver_hier, filt_vert_final);
}
```